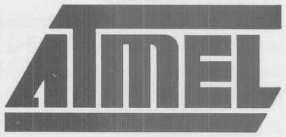


8-Bit Microcontroller with 4 Kbytes Flash

AT89C51 Application Note

Using a Personal Computer to Program the AT89C51

Introduction

This note describes a personal computer-based programmer for the AT89C51 Flash-based Microcontroller. The techniques illustrated here can also be used to implement in-system programming in a user's application environment. All AT89C51 functions are supported, including code read, code write, chip erase, signature read, lock bit write, and Vpp select. Code write, chip erase and lock bit write may be performed at either five or twelve volts, as set by the Vpp select function.

The programmer consists of a hardware unit and its control program. The hardware unit is connected to an IBM PC-compatible host computer through one of the host's parallel ports. Power to the unit is provided by an external power supply. The control program, which was written in Microsoft 'C' (version 5.10), runs on the host.

The emphasis in this design is on ease of implementation. Use of the parallel interface allows a minimum component count. Required delays are enforced by the control program, utilizing software loops. This means that the accuracy of delay intervals will vary between host systems running at different speeds. The code presented here was tested on an 80386-based system running at 33 MHz, and may require modification for use on other systems.

Programming the Flash

The AT89C51 is shipped with the on-chip Flash memory array in the erased state (i.e., contents=FFH) and ready to be programmed. The programming interface accepts either a High Vpp (12V) or a Low Vpp (5V) program enable signal. The Low Vpp programming mode provides a convenient way to program the AT89C51 inside the user's system.

The AT89C51 code memory array is programmed byte-by-byte in either programming mode. To program any non-blank byte in the on-chip Flash Code Memory, the entire memory needs to be erased using the Chip Erase mode.

Hardware Configuration

The hardware unit contains the host interface and circuitry to generate the signals required to control the AT89C51 (see Figure 1). Power for the internal circuitry and the voltages required by the AT89C51 are provided by two external power supplies. One supply is fixed at five volts. A second supply provides either five or twelve volts, selectable by the control program (signal 12VEN*). When power is first applied, all data and control signals are three-stated with Select Vpp set to 5V.

The hardware unit is connected to the host with a 25-conductor cable. The length of the cable should be as short as possible, preferably not exceeding three feet.

Software Control Program

The programmer control program (available on disk from your local Atmel office) is invoked from the DOS command line by entering the program name followed by 'LPT1' or 'LPT2' to select parallel port one or two, respectively. If the parallel port is not specified, the program will prompt for the port. The control program is menu-driven, and provides the following functions:

Select Vpp

Vpp may be set to either five or twelve volts. This function should be selected prior to the code write, chip erase or lock bit write functions.

Chip Erase

Clear code memory to all ones. A blank check should be performed after chip erase to verify the successful completion of this function.

Program from File

Write the contents of the specified file into code memory. The user is prompted for the file name, which may require path and extension. The file size must be 4096 bytes or greater, or an error will occur.

Programming occurs regardless of the existing contents of code memory. After programming, the contents of code memory is not automatically verified against the file data.

The Ready/Busy# signal (at port pin P3.4 of the AT89C51) is available at the Busy terminal of the host's parallel port (pin 11 of the DB25 connector), and can be used to monitor the progress of the self-timed byte write cycle. The program presented here uses a fixed time delay to determine the end of a byte write cycle.

Verify against File

Compare the contents of code memory against the contents of the specified file. The user is prompted for the file name, which may require path and extension. The file size must be 4096 bytes or greater, or an error will occur.

Locations which fail to compare are displayed by address, with the expected and actual byte contents.

Save to File

Copy the contents of code memory to the specified file. The user is prompted for the file name, which may require path and extension. The size of the resulting file is always 4096 bytes.

Blank Check

Verify that the contents of code memory are all ones. Only pass or fail is reported; the addresses of failing locations are not displayed.

Read Signature

Read and display the contents of the two signature bytes, which should be 1EH (ATMEL) and 51H (89C51).

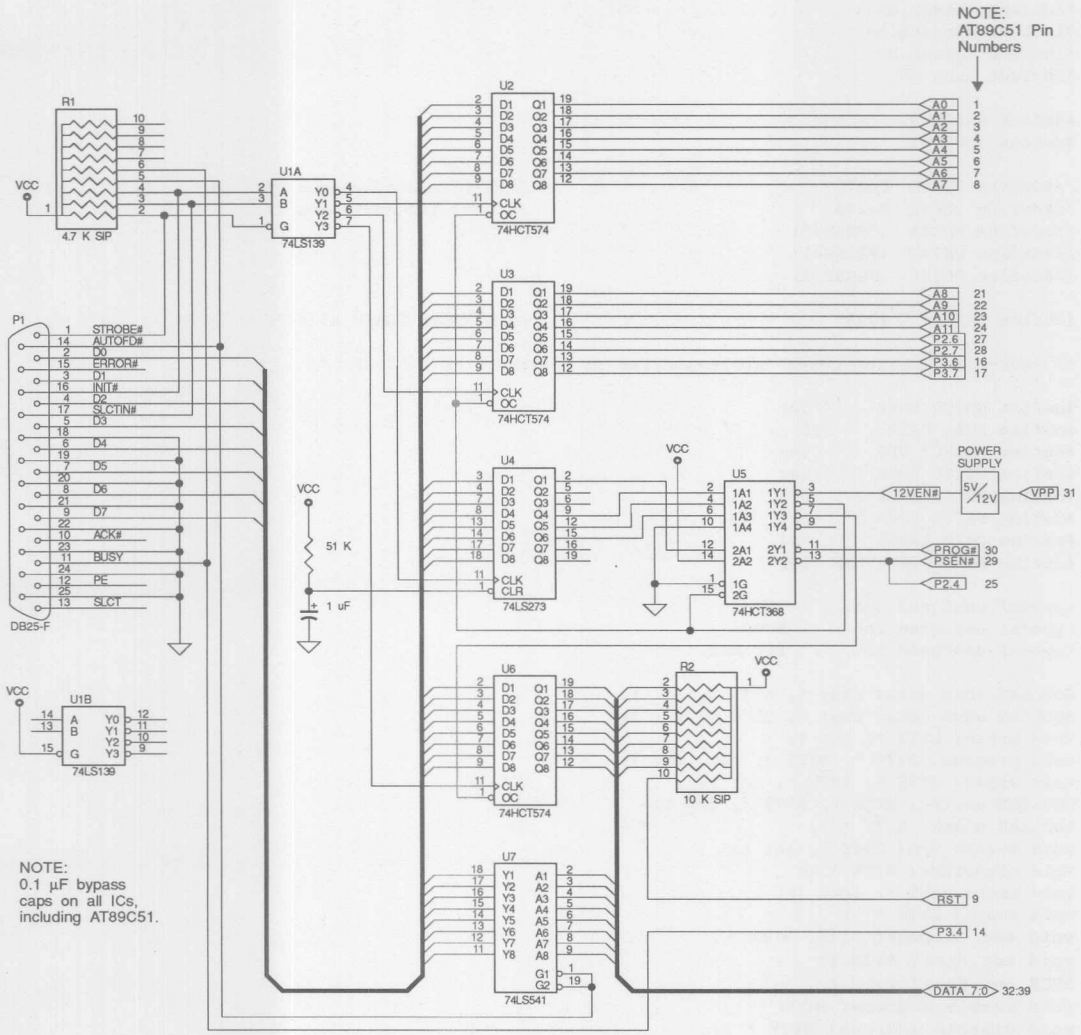
Write Lock Bit 1, Bit 2, Bit 3

Set the indicated lock bit. The state of the lock bits cannot be verified directly. Verification of the lock bits is achieved by observing that the respective protection modes are enabled.

Exit

Quit the programmer control program.

Figure 1. AT89C51 Programmer



AT89C51 Programmer Control Program

```
#include <stdio.h>
#include <string.h>
#include <graph.h>
#include <dos.h>

#define FALSE 0
#define TRUE -1

//#define PBASE  0x378                          /* LPT1 I/O addresses */
//#define PBASE  0x278                          /* LPT2 I/O addresses */
//#define PDATA  (PBASE+0)
//#define PSTAT  (PBASE+1)
//#define PCTRL  (PBASE+2)

#define DSIZE    4096                           /* AT89C51 FLASH size */

/* Four-bit function codes (corresponding to P3.7:P3.6:P2.7:P2.6). */

#define WRITE_DATA      0xe
#define READ_DATA       0xc
#define SELECT_VPP      0xa
#define WRITE_LOCK_1    0xf
#define WRITE_LOCK_2    0x3
#define WRITE_LOCK_3    0x5
#define CHIP_ERASE      0x1
#define READ_SIGNATURE  0x0

typedef unsigned char     BYTE;
typedef unsigned int      WORD;
typedef unsigned int      BOOLEAN;

BOOLEAN load_data( char *, BYTE *, int );
BOOLEAN save_data( char *, BYTE *, int );
void erase( BYTE *, int );
void program( BYTE *, BYTE *, int, int );
void xread( BYTE *, BYTE *, int );
BOOLEAN verify( BYTE *, BYTE *, int );
BOOLEAN blank( BYTE * );
void select_Vpp( BYTE *, int );
void signature( BYTE * );
void lock( BYTE *, int, int );
void reset( BYTE * );
void set_address( BYTE, WORD );
void set_data( BYTE );
BYTE get_data( void );
void enable_address( BYTE * );
void disable_address( BYTE * );
void enable_data( BYTE * );
void disable_data( BYTE * );
void enable_Vpp( BYTE * );
void disable_Vpp( BYTE * );
void slow_pulse( BYTE * );
void fast_pulse( BYTE * );

int pctrl, pdata;                               /* LPTx control and data port addresses */


main(argc, argv)
int argc;
```

```
char *argv[];
{
    FILE *fptr;
    BYTE pgmdata[DSIZE], control = 0;
    char *pch, fname[20];
    WORD far *p_lpt1 = (WORD far *)0x00400008;
    WORD far *p_lpt2 = (WORD far *)0x0040000a;

    if ( (argc > 1) && ((pch = strpbrk( argv[1], "12" )) != NULL) ) {
        switch (*pch) {
        case '1':                                   /* LPT1 */
          pdata = *p_lpt1;
          pctrl = *p_lpt1 + 2;
          break;
        case '2':                                   /* LPT2 */
          pdata = *p_lpt2;
          pctrl = *p_lpt2 + 2;
        }
        if (pdata == 0) {                           /* port undefined */
          puts( "Specified Parallel Port is not implemented." );
          exit( 255 );
        }
    } else {
        puts( "Parallel Port 1 or 2 must be specified on command line." );
        puts( "Usage: <fname> <LPT1 | LPT2>" );
        exit( 255 );
    }

    while ( TRUE ) {
        _clearscreen( _GCLEARSCREEN );

        /*
         *  Menus.
         *  Anything typed on the command line following the port
         *  will invoke the long form menu. All commands are available
         *  with either menu, but only a subset are displayed in the
         *  short form menu.
         */

        if (argc > 2) {                             /* long form */
          puts( "\t\t     5 Volts\t     12 Volts\n" );
          puts( "Select Vpp\t\tA\t\tB\n" );
          puts( "Chip Erase\t\tC\t\tD\n" );
          puts( "Program from File\tE\t\tF" );
          puts( "Verify against File\tG\t\tG" );
          puts( "Save to File\t\tH\t\tH\n" );
          puts( "Blank Check\t\tI\t\tI\n" );
          puts( "Read Signature\t\tJ\t\tJ\n" );
          puts( "Write Lock Bit 1\tK\t\tL" );
          puts( "Write Lock Bit 2\tM\t\tN" );
          puts( "Write Lock Bit 3\tP\t\tQ\n" );
          puts( "Exit\t\t\t\tX\n\n" );
        } else {                                    /* short form */
          puts( "Select Vpp = 5 volts\tA" );
          puts( "Select Vpp = 12 volts\tB\n" );
          puts( "Chip Erase\t\tD\n" );
          puts( "Program from File\tF" );
          puts( "Verify against File\tG" );
          puts( "Save to File\t\tH\n" );
          puts( "Blank Check\t\tI\n" );
          puts( "Read Signature\t\tJ\n" );
          puts( "Write Lock Bit 1\tL" );
          puts( "Write Lock Bit 2\tN" );
```

```
    puts( "Write Lock Bit 3\tQ\n" );
    puts( "Exit\t\t\tX\n\n" );
  }

  printf( "Enter selection: " );
  gets( pch );
  *pch |= 0x20;                              /* convert first char to lower case */

  switch (*pch) {
    case 'a':                                /* select Vpp=5v */
       select_Vpp( &control, 0 );
       break;
    case 'b':                                /* select Vpp=12v */
       select_Vpp( &control, 1 );
       break;
    case 'c':                                /* chip erase @5v */
       erase( &control, 0 );
       break;
    case 'd':                                /* chip erase @12v */
       erase( &control, 1 );
       break;
    case 'e':                                /* write chip from file @5v */
       printf( "Enter file name: " );
       gets( fname );
       if (load_data( fname, pgmdata, DSIZE ))
          program( &control, pgmdata, DSIZE, 0 );
       else {
          _clearscreen( _GCLEARSCREEN );
          puts( "Error opening or reading input data file." );
          puts( "\nPress Enter to continue..." );
          gets( pch );
       }
       break;
    case 'f':                                /* write chip from file @12v */
       printf( "Enter file name: " );
       gets( fname );
       if (load_data( fname, pgmdata, DSIZE ))
          program( &control, pgmdata, DSIZE, 1 );
       else {
          _clearscreen( _GCLEARSCREEN );
          puts( "Error opening or reading input data file." );
          puts( "\nPress Enter to continue..." );
          gets( pch );
       }
       break;
    case 'g':                                /* compare chip contents to file */
       printf( "Enter file name: " );
       gets( fname );
       if (load_data( fname, pgmdata, DSIZE )) {
          if (!verify( &control, pgmdata, DSIZE )) {
             puts( "\nPress Enter to continue..." );
             gets( pch );
          }
       } else {
          _clearscreen( _GCLEARSCREEN );
          puts( "Error opening or reading input data file." );
          puts( "\nPress Enter to continue..." );
          gets( pch );
       }
       break;
    case 'h':                                /* save chip contents to file */
       printf( "Enter file name: " );
       gets( fname );
```

```
            xread( &control, pgmdata, DSIZE );
            if (!save_data( fname, pgmdata, DSIZE )) {
               _clearscreen( _GCLEARSCREEN );
               puts( "Error opening or reading output data file." );
               puts( "\nPress Enter to continue..." );
               gets( pch );
            }
            break;
         case 'i':                                  /* verify blank chip */
            _clearscreen( _GCLEARSCREEN );
            if (blank( &control ))
               puts( "Device is blank" );
            else
               puts( "Device is not blank" );
            puts( "\nPress Enter to continue..." );
            gets( pch );
            break;
         case 'j':                                  /* read signature bytes */
            _clearscreen( _GCLEARSCREEN );
            signature( &control );
            puts( "\nPress Enter to continue..." );
            gets( pch );
            break;
         case 'k':                                  /* write lock bit 1 @5v */
            lock( &control, 1, 0 );
            break;
         case 'l':                                  /* write lock bit 1 @12 v */
            lock( &control, 1, 1 );
            break;
         case 'm':                                  /* write lock bit 2 @5v */
            lock( &control, 2, 0 );
            break;
         case 'n':                                  /* write lock bit 2 @12 v */
            lock( &control, 2, 1 );
            break;
         case 'p':                                  /* write lock bit 3 @5v */
            lock( &control, 3, 0 );
            break;
         case 'q':                                  /* write lock bit 3 @12 v */
            lock( &control, 3, 1 );
            break;
         case 'x':                                  /* exit program */
         default:
            _clearscreen( _GCLEARSCREEN );
            exit( 0 );
      }
   }
}


/*
 * Read data from indicated file into specified array.
 * Returns a boolean value indicating success or failure.
 */
BOOLEAN load_data( fname, store, bcount )
char fname[];
BYTE store[];
int bcount;
{
   FILE *fptr;

   if ((fptr = fopen( fname, "rb" )) == NULL)
      return( FALSE );                              /* file open failed */
```

```
    if (fread( store, bcount, 1, fptr ) != 1)
        return( FALSE );                        /* file read failed */
    fclose( fptr );
    return( TRUE );
}


/*
 * Write data from specified array into indicated file.
 * Returns a boolean value indicating success or failure.
 */
BOOLEAN save_data( fname, store, bcount )
char fname[];
BYTE store[];
int bcount;
{
    FILE *fptr;

    if ((fptr = fopen( fname, "wb" )) == NULL)
        return( FALSE );                        /* file open failed */
    if (fwrite( store, bcount, 1, fptr ) != 1)
        return( FALSE );                        /* file write failed */
    fclose( fptr );
    return( TRUE );
}


/*
 * Clear the chip memory to all ones. Suggested before programming.
 * The erase voltage parameter corresponds to 12 volts if one,
 * five volts if zero.
 */
void erase( cptr, vsel )
BYTE *cptr;
int vsel;
{
    int i;

    reset( cptr );                              /* float signals, Vpp=5v */
    set_address( CHIP_ERASE, 0xfff );           /* select function */
    enable_address( cptr );                     /* enable func, PSEN*, PROG* */
    for (i=0; i<10; i++)                        /* delay */
        ;

    if (vsel) {
        enable_Vpp( cptr ); /* Vpp=12v */
        for (i=0; i<25000; i++)                 /* delay 15 mS Vpp rise->PROG* */
          ;
    }

    slow_pulse( cptr );                         /* apply PROG* pulse */
    for (i=0; i<10; i++)                        /* delay PROG*->addr/data */
        ;
    reset( cptr );                              /* float signals, Vpp=5v */

    if (vsel)
        for (i=0; i<25000; i++)                 /* delay 15 mS for Vpp fall */
          ;
}


/*
 * Program the chip with the contents of the specified data array.
```

```
 * The programming voltage parameter corresponds to 12 volts if one,
 * five volts if zero.
 * The maximum programming time is allotted to each byte write.
 * No attempt is made to determine if a byte programs correctly or
 * if it is already programmed with the desired value.
 */
void program( cptr, data, count, vsel )
BYTE *cptr, data[];
int count, vsel;
{
    WORD addr;
    int i;

    reset( cptr );                                  /* float signals, Vpp=5v */
    set_address( WRITE_DATA, 0xfff );               /* select function */
    enable_address( cptr );                         /* enable func, PSEN*, PROG* */
    set_data( 0xff );
    enable_data( cptr );                            /* enable bus before write */

    if (vsel) {
        for (i=0; i<10; i++)                        /* delay function->Vpp */
          ;
        enable_Vpp( cptr ); /* Vpp=12v */
        for (i=0; i<25000; i++)                     /* delay 15 mS Vpp rise->PROG* */
          ;
    }

    for (addr=0; addr<count; addr++) {
        set_address( WRITE_DATA, addr );            /* apply address */
        set_data( data[addr] );                     /* apply data */
        for (i=0; i<10; i++)                        /* delay addr/data->PROG* */
          ;
        fast_pulse( cptr );                         /* apply PROG* pulse */
        for (i=0; i<10; i++)                        /* delay PROG*->addr/data */
          ;
    }

    reset( cptr );                                  /* float signals, Vpp=5v */

    if (vsel)
        for (i=0; i<25000; i++)                     /* delay 15 mS for Vpp fall */
          ;
}


/*
 * Read the contents of the chip into the specified data array.
 */
void xread( cptr, data, count )
BYTE *cptr, data[];
int count;
{
    BYTE tmp;
    BOOLEAN flag = TRUE;
    WORD addr;
    int i;

    reset( cptr );                                  /* float signals, Vpp=5v */
    set_address( READ_DATA, 0xfff );                /* select function */
    enable_address( cptr );                         /* enable func, PSEN*, PROG* */

    for (addr=0; addr<count; addr++) {
        set_address( READ_DATA, addr );             /* apply address */
```

```
        for (i=0; i<10; i++)                        /* delay address->data */
          ;
        data[addr] = get_data();
    }

    reset( cptr );                                  /* float signals, Vpp=5v */
}


/*
 * Compare the contents of the chip to the specified data array.
 * Failures are displayed by address, with actual contents and expected
 * contents. A boolean value is returned indicating verify pass/fail.
 */
BOOLEAN verify( cptr, data, count )
BYTE *cptr, data[];
int count;
{
    BYTE tmp;
    BOOLEAN flag = TRUE;
    WORD addr;
    int i;

    reset( cptr );                                  /* float signals, Vpp=5v */
    set_address( READ_DATA, 0xfff );                /* select function */
    enable_address( cptr );                         /* enable func, PSEN*, PROG* */

    for (addr=0; addr<count; addr++) {
        set_address( READ_DATA, addr );             /* apply address */
        for (i=0; i<10; i++)                        /* delay address->data */
          ;
        if ((tmp = get_data()) != data[addr]) {
          if (flag) {                               /* first time only */
             _clearscreen( _GCLEARSCREEN );
          }
          printf( "verify fail at %.4X  is %.2X sb %.2X\n", addr, tmp, data[addr] );
          flag = FALSE;
        }
    }

    reset( cptr );                                  /* float signals, Vpp=5v */
    return( flag );
}


/*
 * Determine if the chip is erased. Locations of failures are not
 * determined. A boolean value is returned indicating blank/not blank.
 */
BOOLEAN blank( cptr )
BYTE *cptr;
{
    BYTE tmp;
    BOOLEAN flag = TRUE;                            /* default is blank */
    WORD addr;
    int i;

    reset( cptr );                                  /* float signals, Vpp=5v */
    set_address( READ_DATA, 0xfff );                /* select function */
    enable_address( cptr );                         /* enable func, PSEN*, PROG* */

    for (addr=0; addr<DSIZE; addr++) {
        set_address( READ_DATA, addr );             /* apply address */
```

```
        for (i=0; i<10; i++)                    /* delay address->data */
          ;
        if (get_data() != 0xff)                   /* compare to erased value */
          flag = FALSE;                         /* not blank */
    }

    reset( cptr );                              /* float signals, Vpp=5v */
    return( flag );
}


/*
 * Specify Vpp equal to five volts or 12 volts.
 * The voltage parameter corresponds to 12 volts if one,
 * five volts if zero.
 */
void select_Vpp( cptr, vsel )
BYTE *cptr;
int vsel;
{
    int i;

    reset( cptr );                              /* float signals, Vpp=5v */

    if (vsel) {
        set_address( SELECT_VPP, 0x0aa );       /* select function */
        enable_address( cptr );                 /* enable func, PSEN*, PROG* */
        set_data( 0x55 );
    } else {
        set_address( SELECT_VPP, 0x055 );       /* select function */
        enable_address( cptr );                 /* enable func, PSEN*, PROG* */
        set_data( 0xaa );
    }

    enable_data( cptr );                        /* enable bus before write */
    for (i=0; i<10; i++)                        /* delay function->Vpp */
        ;
    enable_Vpp( cptr );                         /* Vpp=12v */
    for (i=0; i<25000; i++)                     /* delay 15 mS Vpp rise->PROG* */
        ;
    slow_pulse( cptr );                         /* apply PROG* pulse */
    for (i=0; i<10; i++)                        /* delay PROG*->addr/data */
        ;
    reset( cptr );                              /* float signals, Vpp=5v */
    for (i=0; i<25000; i++)                     /* delay 15 mS for Vpp fall */
        ;
}


/*
 * Read signature bytes.
 * The first byte is at address 30h, the second at 31h. When set to
 * 1Eh and 51h, respectively, they identify an AT89C51.
 */
void signature( cptr )
BYTE *cptr;
{
    BYTE tmp1, tmp2;
    int i;

    reset( cptr );                              /* float signals, Vpp=5v */
    set_address( READ_SIGNATURE, 0x030 );       /* select function, address */
    enable_address( cptr );                     /* enable func, PSEN*, PROG* */
```

```
    for (i=0; i<10; i++)                          /* delay address->data */
        ;
    tmp1 = get_data();                            /* read first byte */

    set_address( READ_SIGNATURE, 0x031 );         /* select function */
    for (i=0; i<10; i++)                          /* delay address->data */
        ;
    tmp2 = get_data();                            /* read second byte */

    printf( "signature byte 1:  %.2X\nsignature byte 2:  %.2X\n", tmp1, tmp2 );
    reset( cptr );                                /* float signals, Vpp=5v */
}


/*
 * Write specified lock bit.
 * The voltage parameter corresponds to 12 volts if one,
 * five volts if zero.
 */
void lock( cptr, lbit, vsel )
BYTE *cptr;
int lbit, vsel;
{
    int i;

    reset( cptr );                                /* float signals, Vpp=5v */

    switch (lbit) {                               /* select function */.
        case 1:
          set_address( WRITE_LOCK_1, 0xfff );
          break;
        case 2:
          set_address( WRITE_LOCK_2, 0xfff );
          break;
        case 3:
          set_address( WRITE_LOCK_3, 0xfff );
          break;
    }

    enable_address( cptr );                       /* enable func, PSEN*, PROG* */
    for (i=0; i<10; i++)                          /* delay */
        ;

    if (vsel) {
        enable_Vpp( cptr ); /* Vpp=12v */
        for (i=0; i<25000; i++)                   /* delay 15 mS Vpp rise->PROG* */
          ;
    }

    fast_pulse( cptr );                           /* apply PROG* pulse */
    for (i=0; i<10; i++)                          /* delay PROG*->addr/data */
        ;
    reset( cptr );                                /* float signals, Vpp=5v */

    if (vsel)
        for (i=0; i<25000; i++)                   /* delay 15 mS for Vpp fall */
          ;
}


/*
 * Return programmer hardware to the passive state.
 * Writes zeros into the programmer control latch, which floats the
```

```
 * address and data busses and puts 5 volts on Vpp. The programmer
 * address latches are loaded with zeros, the data latch with ones.
 * NOTE: The programmer will not power up correctly with ones left
 * on the LPTx data bus. Also, latches must be addressed before
 * the strobe is generated to avoid glitches.
 */
void reset( cptr )
BYTE *cptr;
{
   outp( pdata, 0 );                          /* set up data */

   outp( pctrl, 0x08 );                       /* select control latch */
   outp( pctrl, 0x09 );                       /* latch data */
   outp( pctrl, 0x08 );

   outp( pctrl, 0x0c );                       /* select low address latch */
   outp( pctrl, 0x0d );                       /* latch data */
   outp( pctrl, 0x0c );

   outp( pctrl, 0x00 );                       /* select high address latch */
   outp( pctrl, 0x01 );                       /* latch data */
   outp( pctrl, 0x00 );

   outp( pdata, 0xff );                       /* set up data */

   outp( pctrl, 0x04 );                       /* select data latch */
   outp( pctrl, 0x05 );                       /* latch data */
   outp( pctrl, 0x04 );

   outp( pdata, 0 );                          /* data inactive */
   outp( pctrl, 0x08 );                       /* control signals inactive */

   *cptr = 0;                                 /* save control latch value */
}


/*
 * Write specified values into the programmer address and function
 * code latches. Both address and function must be specified.
 */
void set_address( func, addr )
BYTE func;
WORD addr;
{
   outp( pdata, addr );                       /* set up low byte of address */

   outp( pctrl, 0x0c );                       /* select low address latch */
   outp( pctrl, 0x0d );                       /* latch data */
   outp( pctrl, 0x0c );

   outp( pdata, (func << 4) | (addr >> 8) );
                                              /* set up high byte of address */

   outp( pctrl, 0x00 );                       /* select high address latch */
   outp( pctrl, 0x01 );                       /* latch data */
   outp( pctrl, 0x00 );

   outp( pdata, 0 );                          /* clear data */
   outp( pctrl, 0x08 );                       /* control signals inactive */
}
```

```
/*
 * Write specified value into the programmer data latch.
 */
void set_data( outdata )
BYTE outdata;
{
    outp( pdata, outdata );                     /* set up output data */

    outp( pctrl, 0x04 );                        /* select data latch */
    outp( pctrl, 0x05 );                        /* latch data */
    outp( pctrl, 0x04 );

    outp( pdata, 0 );                           /* clear data */
    outp( pctrl, 0x08 );                        /* control signals inactive */
}


/*
 * Return data present on programmer data lines.
 * Does not first disable programmer output data latch.
 */
BYTE get_data( void )
{
    BYTE tmp;
    int i;

    outp( pdata, 0xff );                        /* set LPTx port data high */

    outp( pctrl, 0x02 );                        /* enable read data buffer */
    for (i=0; i<10; i++)                        /* delay */
        ;
    tmp = inp( pdata );                         /* get data */

    outp( pdata, 0 );                           /* clear data */
    outp( pctrl, 0x08 );                        /* control signals inactive */

    return( tmp );
}


/*
 * Enable outputs of programmer address latches.
 * Note that PSEN* and PROG* are also enabled.
 */
void enable_address( cptr )
BYTE *cptr;
{
    outp( pdata, (*cptr |= 0x10) );             /* set up data */

    outp( pctrl, 0x08 );                        /* select control latch */
    outp( pctrl, 0x09 );                        /* latch data */
    outp( pctrl, 0x08 );

    outp( pdata, 0 );                           /* clear data */
// outp( pctrl, 0x08 );                         /* control signals inactive */
}


/*
 * Disable outputs of programmer address latches.
 * Note that PSEN* and PROG* are also disabled.
 */
void disable_address( cptr )
```

```
BYTE *cptr;
{
    outp( pdata, (*cptr &= ~0x10) );              /* set up data */

    outp( pctrl, 0x08 );                          /* select control latch */
    outp( pctrl, 0x09 );                          /* latch data */
    outp( pctrl, 0x08 );

    outp( pdata, 0 );                             /* clear data */
// outp( pctrl, 0x08 );                           /* control signals inactive */
}


/*
 * Enable output of programmer data latch.
*/
void enable_data( cptr )
BYTE *cptr;
{
    outp( pdata, (*cptr |= 0x20) );               /* set up data */

    outp( pctrl, 0x08 );                          /* select control latch */
    outp( pctrl, 0x09 );                          /* latch data */
    outp( pctrl, 0x08 );

    outp( pdata, 0 );                             /* clear data */
// outp( pctrl, 0x08 );                           /* control signals inactive */
}


/*
 * Disable output of programmer data latch.
 */
void disable_data( cptr )
BYTE *cptr;
{
    outp( pdata, (*cptr &= ~0x20) );              /* set up data */

    outp( pctrl, 0x08 );                          /* select control latch */
    outp( pctrl, 0x09 );                          /* latch data */
    outp( pctrl, 0x08 );

    outp( pdata, 0 );                             /* clear data */
// outp( pctrl, 0x08 );                           /* control signals inactive */
}


/*
 * Enable 12 volts on Vpp.
 * Note that Vpp will not immediately reach the specified value.
 */
void enable_Vpp( cptr )
BYTE *cptr;
{
    outp( pdata, (*cptr |= 0x08) );               /* set up data */

    outp( pctrl, 0x08 );                          /* select control latch */
    outp( pctrl, 0x09 );                          /* latch data */
    outp( pctrl, 0x08 );

    outp( pdata, 0 );                             /* clear data */
// outp( pctrl, 0x08 );                           /* control signals inactive */
}
```

```
/*
 * Return Vpp to 5 volts.
 * Note that Vpp will not immediately reach the specified value.
 */
void disable_Vpp( cptr )
BYTE *cptr;
{
   outp( pdata, (*cptr &= ~0x08) );              /* set up data */

   outp( pctrl, 0x08 );                          /* select control latch */
   outp( pctrl, 0x09 );                          /* latch data */
   outp( pctrl, 0x08 );

   outp( pdata, 0 );                             /* clear data */
// outp( pctrl, 0x08 );                          /* control signals inactive */
}


/*
 * Generate 10 mS low-going pulse on PROG*.
 * Note that the software timing loop is system-dependent.
 */
void slow_pulse( cptr )
BYTE *cptr;
{
   int i;

   outp( pdata, (*cptr |= 0x01) );               /* set up data */

   outp( pctrl, 0x08 );                          /* select control latch */
   outp( pctrl, 0x09 );                          /* latch data */
   outp( pctrl, 0x08 );

   for ( i=0; i<15000; i++ )                     /* approx 10 mS on 33 MHz 80386 */
        ;        /* delay */

   outp( pdata, (*cptr &= ~0x01) );              /* set up data */

   outp( pctrl, 0x08 );                          /* select control latch */
   outp( pctrl, 0x09 );                          /* latch data */
   outp( pctrl, 0x08 );

   outp( pdata, 0 );                             /* clear data */
// outp( pctrl, 0x08 );                          /* control signals inactive */
}


/*
 * Generate 100 uS low-going pulse on PROG*.
 * Note that the software timing loop is system-dependent.
 */
void fast_pulse( cptr )
BYTE *cptr;
{
   int i;

   outp( pdata, (*cptr |= 0x01) );               /* set up data */

   outp( pctrl, 0x08 );                          /* select control latch */
   outp( pctrl, 0x09 );                          /* latch data */
   outp( pctrl, 0x08 );
```

```
    for ( i=0; i<150; i++ )                    /* approx 100 uS on 33 MHz 80386 */
        ;       /* delay */

    outp( pdata, (*cptr &= ~0x01) );           /* set up data */

    outp( pctrl, 0x08 );                       /* select control latch */
    outp( pctrl, 0x09 );                       /* latch data */
    outp( pctrl, 0x08 );

    outp( pdata, 0 );                          /* clear data */
//  outp( pctrl, 0x08 );                       /* control signals inactive */
}
```